DESIGN AND IMPLEMENTATION OF A JAVA SOAP TOOLKIT

YANG YU

A MAJOR REPORT
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

MARCH, 2003

Abstract

Design and Implementation of a Java SOAP Toolkit

Yang Yu

As the underlying transport protocol of Web Services, Simple Object Access Protocol (SOAP) is a lightweight protocol for exchange of information in a decentralized, distributed environment. This paper illustrates the design and implementation of a Java SOAP Toolkit (JSTK) which conforms to SOAP specification 1.1 and provides developers a series of simplified Application Programming Interfaces (APIs) to access and deploy a SOAP service on both client and server sides.

Acknowledgements

I would like to thank my supervisor, Prof. Stan Klasa, for his patience and valuable guidance. Without his encouragement and support, this work could not be done.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

1. Introduction of SOAP Protocol 1
1.1 Web Services and SOAP Protocol 1
1.2 SOAP Envelope 1
1.3 SOAP Messaging Model 2
1.4 SOAP/RPC over HTTP 4

2. Java/XML Mapping 9
2.1 Java to XML Mapping 9
2.1.1 Java Primitive Type 9
2.1.2 Standard Java Classes 9
2.1.3 Array of Bytes 10
2.1.4 Java Array 10
2.2 XML to Java Mapping 13
2.2.1 Simple Types 13
2.2.2 Array 13
2.2.3 XML Structure Type 14
2.2.4 Enumeration 16

3. Programming Model 20
3.1 Client-side Programming Model 20
3.2 Service-side programming Model 20

4. Toolkit Architecture 29
4.1 Architecture Overview 29
4.2 Core Packages Design 29
4.2.1 Call Package 31
4.2.2 Encoding Package 31
4.2.3 Transport Package 32
4.2.4 Message Package 33
4.2.5 Provider Package 34

5. Core Packages Implementation 36
5.1 Classes In Call Package 36
5.2 Classes In Encoding Package 37
5.3 Classes In Message Package 37
5.4 Classes In Transport Package 41
5.5 Classes In Provider package 41

6. Conclusion and Future Work 43
6.1 Conclusion 43
6.2 Future Work 44

REFERENCES 45

LIST OF TABLES

Table 1: An Envelope of SOAP Request 3

Table 2: An Envelope of SOAP Response 4

Table 3: An Envelope of SOAP Fault 4

Table 4: A HTTP Request of SOAP/RPC 8

Table 5: A HTTP response of SOAP/RPC 8

Table 6: Mapping of Java Primitive Types 9

Table 7: Mapping of Standard Java Classes 10

Table 8: Java Mapping for Simple XML Data Types 13

Table 9: Java Mapping for Simple XML Types With "nillable" Set To True 14

LIST OF FIGURES

Figure 1: Enterprise Information System Integration with Web Services 1

Figure 2: Model of Simple Case without Intermediaries 5

Figure 3: Model of Complex Case with Intermediaries 6

Figure 4: The Architecture of JSTK Toolkit 29

Figure 5: Sequence Diagram of Making an Invocation on Client Side 30

Figure 6: Sequence Diagram of Processing a SOAP Request on Server Side 31

Figure 7: Class Diagram of the Call Package 32

Figure 8: Encoding Class Diagram 32

Figure 9: Decoding Class Diagram 33

Figure 10: Class Diagram of the Message Package 34

Figure 11: Class Diagram of the Transport Package 34

Figure 12: Class Diagram of the Provider Package 35

Chapter 1

Introduction of SOAP Protocol

1.1 Web Services and SOAP Protocol

The use of Web services on the World Wide Web is expanding rapidly as the need for application-to-application communication and interoperability grows. These services provide a standard means of communication among different software applications involved in presenting dynamic context-driven information to the user [1]. Web Services is a new technology enabling information systems interactively to exchange data over Internet or Intranet in a manner of platform-independence, data-independence and protocol-independence. Web Services is able to minimize the cost of business system integration across enterprises and businesses due to employing non-proprietary data format (Extensible Markup Language(XML) [2]), non-proprietary application protocol (SOAP [3]) and standardized Internet protocols (Hypertext Transfer Protocol(HTTP) [4], Simple Mail Transfer Protocol(SMTP) etc.). A scenario of employing Web Services is depicted as follows:

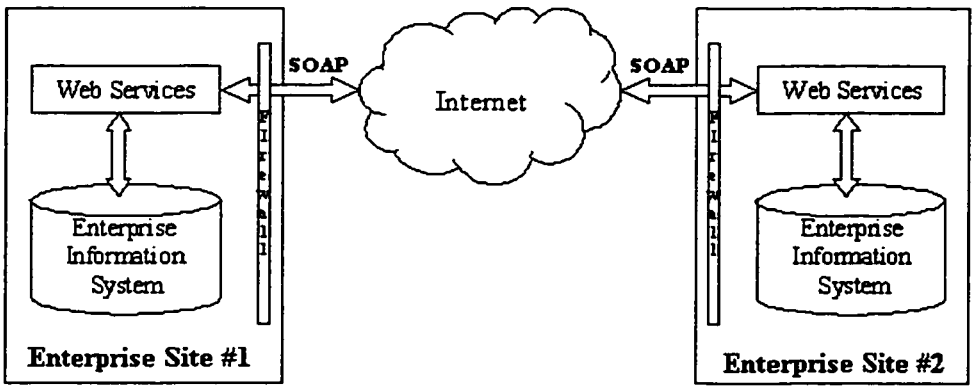


Figure 1: Enterprise Information System Integration with Web Services

After Enterprise #1 becomes the business partner of Enterprise #2, they both attempt to share and exchange their business data. With the traditional technologies, the cost of system integration is extremely high since each enterprise system has a distinct infrastructure, different data representation formats and data access interface. With web services, enterprise system functionality can be exposed via web services in a manner of technology-independence, making enterprise system integration much easier.

As the underlying transport protocol of web services, Simple Object Access Protocol (SOAP) is a lightweight protocol for exchange of information in a decentralized, distributed environment [3]. SOAP is a XML-based protocol providing a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment by using XML. Instead of defining any application semantics, such as a programming model or implementation specific semantics, SOAP defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules, which allows it to be used in a large variety of systems ranging from messaging systems to Remote Procedure Call (RPC). In addition, SOAP is not bound to a specific Internet protocol and can be used in combination with a variety of existing Internet protocols and formats including HTTP, SMTP, and Multipurpose Internet Mail Extensions (MIME).

1.2 SOAP Envelope

A SOAP message is encapsulated in a XML document of so-called "SOAP Envelope" which consists of an optional SOAP header and a mandatory SOAP body. Encoded as the first immediate child element of the SOAP Envelope XML element, SOAP header is used

to provide a flexible mechanism for extending a message in a decentralized and modular way without prior knowledge between the communicating parties. Encoded as an immediate child element of the SOAP Envelope XML element, SOAP body is used to provide a simple mechanism for exchanging mandatory information intended for the ultimate recipient of the message. As an example of SOAP envelope, a request of the SOAP service "EchoString", which echoes a string it has received, is depicted as follows:

```
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance/">
   <SOAP-ENV:Header>
      <a:Authentication xmlns:a="http://www.cs.concordia.ca/jstk/echo/"
                        SOAP-ENV:mustUnderstand="1">
      <a:User>Mike</a:User>
      <a:Password>XXXXXXX<a:Password>
      </a:Authentication>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <m:Echo xmlns:m="http://www.cs.concordia.ca/jstk/echo/">
      <greeting>salute!</greeting>
      </m:Echo>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Table 1: An Envelope of SOAP Request

In the above SOAP envelope, there is a SOAP header used to authenticate the users for the SOAP service "EchoString". The receiver of this envelope must retrieve and process the header in this envelope since the header attribute "mustUnderstand" equals one. That is, only the authenticated users are able to access the SOAP service "EchoString". In the envelope, there is also a SOAP body used to specify a requested operation "Echo" provided by the SOAP service "EchoString". The operation "Echo" has an input parameter "greeting" with the value of "salute!". The corresponding response is specified as follows:

```
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance/">
   <SOAP-ENV:Body>
       <m:EchoResponse xmlns:m="http://www.cs.concordia.ca/jstk/echo/">
       <Result>salute!<Result>
       </m:Echo>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Table 2: An Envelope of SOAP Response

If an error occurs during invoking the service "EchoString", a SOAP fault is about to be created as below:

```
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance/">
   <SOAP-ENV:Body>
       < SOAP-ENV:Fault>
       < SOAP-ENV:faultCode> SOAP-ENV:Receiver</ SOAP-ENV:faultCode>
       < SOAP-ENV:faultString> Processing Error</ SOAP-ENV:faultString>
       </ SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Table 3: An Envelope of SOAP Fault

1.3 SOAP Messaging Model

Depending upon SOAP message path from sender to receiver, SOAP messaging model can be divided into two categories: One-way Transmission and Multi-intermediary Transmission. One –way transmission refers that SOAP message is directly transferred to its ultimate destination; Multi-intermediary transmission refers that SOAP message is transferred to its destination through one or more intermediate nodes. As depicted below, the figure 2.1 is a typical scenario of one-way transmission. The SOAP application one in

the host "I" sends a SOAP message to the SOAP application two in the host "II". The application one invokes a SOAP processor to transfer the message, and the SOAP processor is bound to an underlying protocol(e.g. HTTP, SMTP) to directly send the message to the host "II". After receiving the message via the underlying protocol, the SOAP processor transfers the message to the SOAP application two.

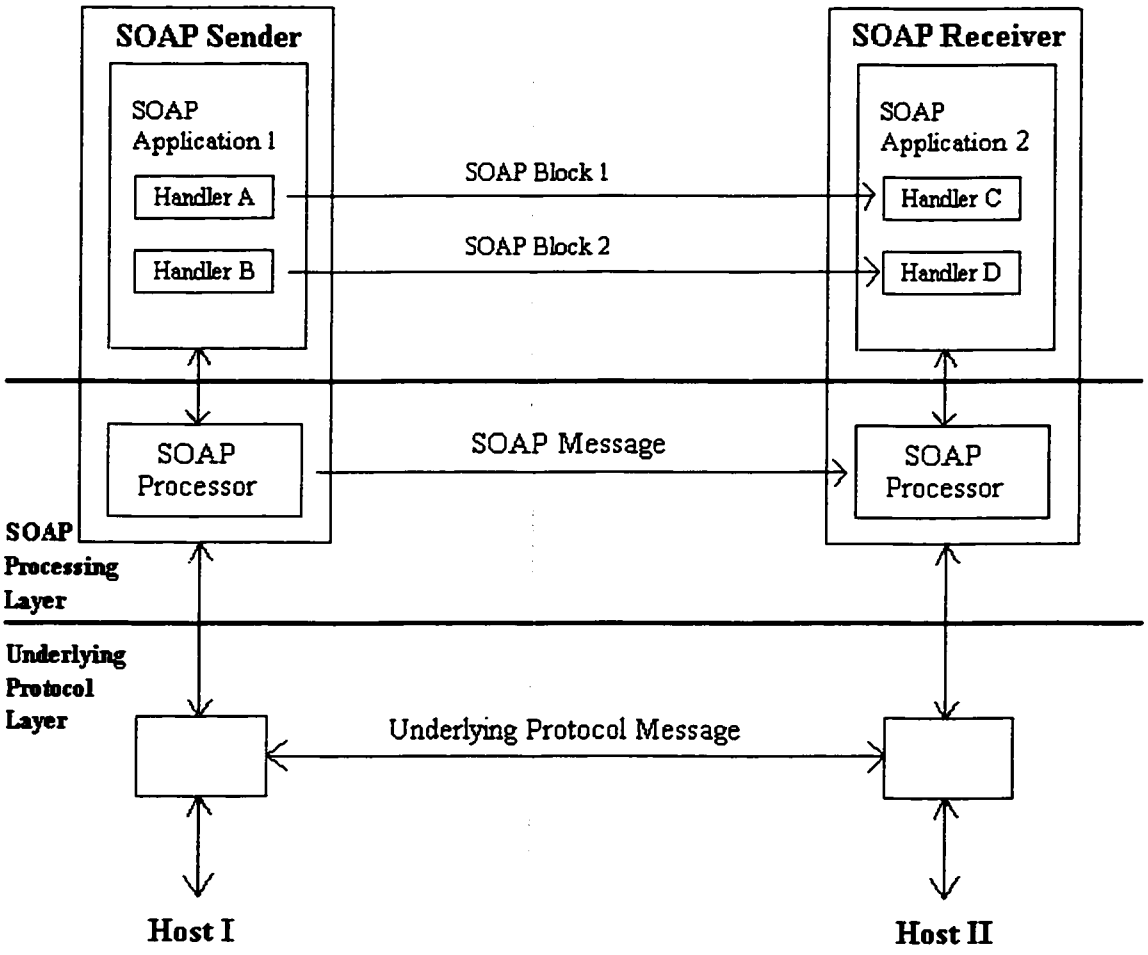


Figure 2: Model of Simple Case without Intermediaries [5]

A SOAP message can also go through one or more intermediate nodes as the figure 2.2 depicts. In case of the host "II" as a HTTP proxy or a SMTP reply, SOAP message is about to go through the host "II" without any SOAP processing. In case of the host "II" as a SOAP intermediator, SOAP message is going to be processed during going through the host "II" and the underlying protocol used between Host I and Host II may be different from the protocol between Host II and Host III.

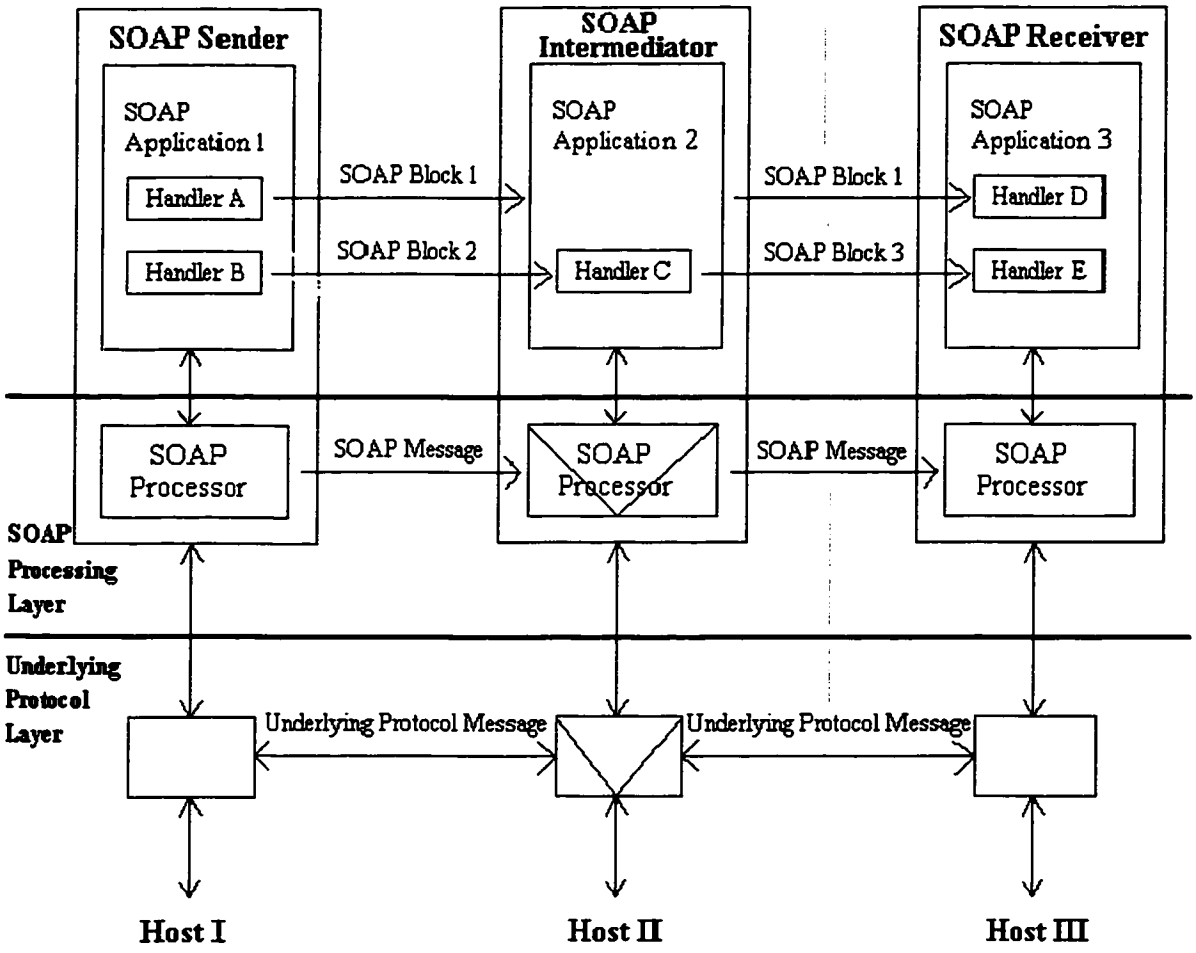


Figure 3: Model of Complex Case with Intermediaries [5]

1.4 SOAP/RPC over HTTP

Although SOAP can be employed in many fields of applications, its primary application is in the field of RPC. The RPC mechanism enables a remote procedure call from a client to be communicated to a remote server. An example use of an RPC mechanism is in a distributed client/server model. A server defines a service as a collection of procedures that are callable by remote clients. A client calls procedures to access service defined by the server. With the traditional technologies like Remote Method Invocation (RMI), Distributed Component Object Model (DCOM) and Common Object Request Broker Architecture (CORBA), a RPC call can not go through the firewalls which most enterprises and organizations have already set up. Since HTTP protocol has the capability of crossing firewalls, binding SOAP to HTTP enables a SOAP/RPC call to go through firewalls, making system integration between enterprises and organizations much easier and quicker. As an example of SOAP/RPC over HTTP, a HTTP request of the SOAP service "EchoString" are depicted as below:

```
POST /jstk/echo HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: 517
SOAPAction: "Echo"


<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance/">
   <SOAP-ENV:Header>
      <a:Authentication xmlns:a="http://www.cs.concordia.ca/jstk/echo/"
                        SOAP-ENV:mustUnderstand="1">
      <a:User>Mike</a:User>
      <a:Password>XXXXXXX<a:Password>
      </a:Authentication>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
```

```
        <m:Echo xmlns:m="http://www.cs.concordia.ca/jstk/echo/">
        <greeting>salute!</greeting>
        </m:Echo>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Table 4: A HTTP Request of SOAP/RPC

The corresponding HTTP response is depicted as follows:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 322


<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance/">
    <SOAP-ENV:Body>
        <m:EchoResponse xmlns:m="http://www.cs.concordia.ca/jstk/echo/">
        <Result>salute!<Result>
        </m:Echo>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Table 5: A HTTP response of SOAP/RPC

Chapter 2

Java/XML Mapping

This chapter specifies a standard mapping between Java [6] and XML since all data types in SOAP message are represented in XML.

2.1 Java to XML Mapping

2.1.1 Java Primitive Type

The following table specifies the standard mapping of the Java primitive types to the XML data types:

Java Primitive Type	**XML Data Type**
Boolean	xsd:boolean
Byte	xsd:byte
Short	xsd:short
Int	xsd:int
Long	xsd:long
Float	xsd:float
Double	xsd:double

Table 6: Mapping of Java Primitive Types

As an example of Java primitive type to XML mapping, "int credits = 3" is supposed to be mapped to the following XML segment:

<credits xsi:type="xsd:int">3</credits>

In the case of literal element declarations, the Java class for a Java primitive type (example: java.lang.Integer) is mapped to an element declaration with the nillableattribute set to true. The following example shows the mapping for the java.lang.Integer:

<credits xsi:type="xsd:int" xsi:nillable ="true">3</credits>

2.1.2 Standard Java Classes

Some of standard Java Classes are mapped to the corresponding XML types in the following table:

Java Class	**XML Data Type**
Java.lang.String	xsd:string
Java.math.BigInteger	xsd:integer
Java.math.BigDecimal	xsd:decimal
Java.util.Calendar	xsd:dateTime
Java.util.Date	xsd:dateTime

Table 7: Mapping of Standard Java Classes

2.1.3 Array of Bytes

Since Java array of bytes represents binary data to be transferred in a XML document, both byte[] and Byte[] are mapped to the xsd:base64Binary type which uses the Base64 algorithm to encode Java array of bytes.

2.1.4 Java Array

According to SOAP specification 1.1, a Java array is mapped to an array with the "soapenc:arrayType" attribute in the XML schema instance as specified in the SOAP 1.1 encoding.

Example 1:

```
Int[] scores = new int[] { 90, 75, 66, 87, 83 };
<scores soapenc:arrayType ="xsd:int">
    <element1>90</element1>
    <element2>75</element2>
    <element3>66</element3>
    <element4>87</element4>
    <element5>84</element5>
</scores>
```

Example 2:

```
Integer[] scores = new Integer[] { new Integer("88"), new Integer("99")};
<scores soapenc:arrayType ="xsd:int" xsi:nillable ="true">
    <element1>88</element1>
    <element2>99</element2>
</scores>
```

2.1.5 Java/RPC Value Type

A Java/RPC value type is a Java class whose value can be moved between a service client and service endpoint. A Java class must follow these rules to be a Java/RPC

conformant value type:

- Java class has no default class constructor
- The member properties of Java class must be public
- Java class has no any operations declared
- Java class must not inherit any other Java class

A Java/RPC value type is mapped to an "xsd:complexType" , and the name attribute of the "xsd:complexType" is mapped from the name of the Java class for the Java/RPC value type.

Example:

```
public class Student {
    public String firstName = "Mike";
    public String lastName = "Amar";
    public boolean isMale = "true";
    public int[] grades = new int[] { 76, 69, 99, 82 };
}
Student student = new Student();
```

```
<student xsi:type = "xsd:complexType" name = "Student">
    <firstName xsi:type = "xsd:string">Mike</firstName>
    <lastName xsi:type = "xsd:string">Amar</lastName>
    <isMale xsi:type = "xsd:boolean">true</isMale>
    <grades soapenc:arrayType ="xsd:int">
        <element1>76</element1>
```

```
        <element2>69</element2>
        <element3>99</element3>
        <element4>82</element4>
    </grades>
</student>
```

2.2 XML to Java Mapping

2.2.1 Simple Types

The following table specifies the Java mapping for the built-in simple XML data types which are defined in the XML schema specification [7].

Simple Type	Java Type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	Int
xsd:long	Long
xsd:short	Short
xsd:decimal	java.math.BigDecimal
xsd:float	Float
xsd:double	Double
xsd:boolean	Boolean
xsd:byte	Byte
xsd:Qname	javax.xml.namespace.QName
xsd:dateTime	java.util.Calendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]

Table 8: Java Mapping for Simple XML Data Types

Example:

```
<stocks xsi:type = "xsd:int">35</stocks>
```

is mapped to:

int stocks = 35;

If simple XML data types have the attribute "nillable" set to true, the XML data types are supposed to be mapped to the corresponding classes of Java primitive types as below:

Simple XML Type	Java Type
xsd:int	java.lang.Integer
xsd:long	java.lang.Long
xsd:short	java.lang.Short
xsd:float	java.lang.Float
xsd:double	java.lang.Double
xsd:boolean	java.lang.Boolean
xsd:byte	java.lang.Byte

Table 9: Java Mapping for Simple XML Types With "nillable" Set To True

Example:

<stocks xsi:type = "xsd:int" xsi:nillable = "true">35</stocks>

is mapped to:

Integer stocks = new Integer("35");

2.2.2 Array

A XML array declared as "soapenc:Array" is mapped to the corresponding Java class. Note that the array dimension is omitted in the declaration of a Java array. The number of elements in a Java array is determined at the creation time rather than when an array is declared.

Example 1:

<maillist soapenc:arrayType ="xsd:string">

<element1>scotie@mcgrill.ca</element1>

<element2>yang@concordia.ca</element2>

</maillist>

is mapped to:

String[] maillist;

Example 2:

<studentlist soapenc:arrayType =”Student”>

<element1>

<firstName xsi:type = "xsd:string">John</firstName>

<lastName xsi:type = "xsd:string">Paton</lastName>

<isMale xsi:type = "xsd:boolean">true</isMale>

<grade xsi:type = "xsd:int">80</grade>

</element1>

<element2>

<firstName xsi:type = "xsd:string">Lisa</firstName>

<lastName xsi:type = "xsd:string">Smith</lastName>

<isMale xsi:type = "xsd:boolean">false</isMale>

<grade xsi:type = "xsd:int">91</grade>

</element2>

</studentlist>

is mapped to:

Student[] studentlist;

Public class Student {

```
    Public String firstName;
    Public String lastName;
    Public boolean isMale;
    Public int grade;
}
```

2.2.3 XML Structure Type

An XML structure (XML struct) is mapped to a public class with the same name as the type of the XML struct. The mapped Java class provides a corresponding public class field for each property mapped from the member elements of the XML struct. The identifier and Java type of a property in the Java class is mapped from the name and type of the corresponding member element in the XML struct.

Example:

```
<textbook xsi:type = "xsd:complexType" name = "Book">
    <authors soapenc:arrayType ="Author">
        <element1>
            <firstName xsi:type = "xsd:string">Grady</firstName>
            <lastName xsi:type = "xsd:string"> Booch</lastName>
            <contact xsi:type = "xsd:string">grady@rational.com</contact>
        </element1>
        <element2>
            <firstName xsi:type = "xsd:string">Ivar</firstName>
            <lastName xsi:type = "xsd:string">Jacobson</lastName>
```

```
            <contact xsi:type = "xsd:string">ivar@hotmail.com</contact>
        </element2>
    </authors>
    <preface xsi:type = "xsd:string">this is the preface of the book.</preface>
    <price xsi:type = "xsd:float">78.00</price>
</textbook>
```

is mapped to:

```
Book textbook;
Public class book {
    Public Author[] authors;
    Public String preface;
    Public float price;
}
public class Author {
    public String firstName;
    public String lastName;
    public String contact;
}
```

2.2.4 Enumeration

An XML enumeration is a specific list of distinct values appropriate for a base type. In JSTK, an XML enumeration is mapped to a Java class which declares a public constant for each enumeration member, and has a constructor with an enumeration member as parameter.

Example:
<today Name = "Weekday">Sunday</today>

is mapped to:

```
Weekday today = Weekday.Sunday;

Public class Weekday {

    private static String _day;

    Private static final String _Monday = "Monday";

    Private static final String _Tuesday = "Tuesday";

    Private static final String _Wednesday = "Wednesday";

    Private static final String _Thursday = "Thursday";

    Private static final String _Friday = "Friday";

    Private static final String _Saturday = "Saturday";

    Private static final String _Sunday = "Sunday";

    Public static final Weekday Monday = new Weekday(_Monday);

    Public static final Weekday Tuesday = new Weekday(_Tuesday);

    Public static final Weekday Wednesday = new Weekday(_Wednesday);

    Public static final Weekday Thursday = new Weekday(_Thursday);

    Public static final Weekday Friday = new Weekday(_Friday);

    Public static final Weekday Saturday = new Weekday(_Saturday);

    Public static final Weekday Sunday = new Weekday(_Sunday);

    Protected Weekday (String day) {

        _day = day;

    }

    public String getValue() {
```

```
        return _day;
    }
    boolean equals(Weekday wDay) {
        if(_day.equals(wDay.getValue()))
          return true;
        else
          return false;
    }
}
```

Chapter 3

Programming Model

This chapter specifies the JSTK's APIs for both client side and server side, and how developers use these APIs to invoke and deploy a SOAP service.

3.1 Client-side Programming Model

JSTK uses a dynamic invocation interface on client side to invoke a SOAP service. The invocation interface is completely independent of how a SOAP service is realized on the server side, moreover, it is not bound to a specific underlying transport protocol. The client-side invocation interface of JSTK is specified in the interface "Call" which is depicted as follows:

```
public interface Call {
   public void addParameter(Parameter param);
   public void addParameter(String paramName, Class paramType, Object paramValue);
   public void removeParameter(String paramName);
   public removeAllParameters();
   public setParameters(Parameters params);
   pubic Parameters getParameters();
   public String getOperationName();
   public void setOperationName(String operationName);
   public void setTargetServiceAddress(URL targetServiceAddress);
   public URL getTargetServiceAddress();
   public void setProperty(String name, Object value);
```

```
    public Object getProperty(String name);
    public boolean removeProperty(String name);
   // Remote Invocation Method
   public Object invoke(String operationName, Parameters inputParams)
            throws SoapException;
   public Object invoke(Parameters inputParams) throws SoapException;
   public Object invoke(String operationName, Object[] inputParams)
            throws SoapException;
   public Object invoke(Object[] inputParams) throws SoapException;
}
```

Developers can use well-named or anonymous parameters via the "Call" interface to invoke a SOAP service, and can also transfer application-level information by using set/get Property methods which are actually transferred in the header of SOAP envelope.The other classes in JSTK related to the interface "Call" are specified as follows:

```
public class Parameters {
    public void add(Parameter para);
    public void removeAll();
    public int getCount();
    public Parameter getParameter(int index);
}

public class Parameter {
    public Parameter(String name, Class type, Object value);
    public void setName(String name);
```

```
    public String getName();
    public void setType(Class type);
    public Class getType();
    public void setValue(Object value);
    public Object getValue();
}
public class SimpleType  {
     public static Class INTEGER = int.class;
     public static Class BOOLEAN = boolean.class;
     public static Class BYTE = byte.class;
     public static Class SHORT = short.class;
     public static Class DOUBLE = double.class;
     public static Class LONG = long.class;
     public static Class FLOAT = float.class;
     public static Class STRING = new String().getClass();
     public static Class BIG_INTEGER = new BigInteger("0").getClass();
     public static Class DECIMAL = new BigDecimal("0").getClass();
     public static Class DATE = new Date().getClass();
}
```

The following examples show how to use the interface "Call" and its related classes to invoke a specific SOAP service:

Example 1:

Suppose there is a SOAP service called "EchoString" deployed at http://www.cs.concordia.ca/jstk/echo, and the SOAP service has a method "echo" with the signature "String echo(String str)", the code snippet of invoking this service is illustrated as below:

```
……..
Call call = new Call();
call.setTargetServiceAddress(new URL("http://www.cs.concordia.ca/jstk/echo"));
call.setOperationName("echo");
String result = (String)call.invoke(new Object[]{"Bonjour!"});
……..
```

In the above example, anonymous parameters which have no specified names are used to invoke the SOAP service. Additionally, JSTK's invocation interface also provides a means to invoke a SOAP service by using well-named parameters as depicted in below:

Example 2:

Suppose there is a SOAP service called "Calculator" deployed at http://www.cs.concordia.ca/jstk/calculator, and the SOAP service has a method "sum" with the signature "int sum(int num1, int num2)", the code snippet of invoking this service is illustrated as below:

```
……..
Call call = new Call();
URL servAddr = new URL("http://www.cs.concordia.ca/jstk/calculator");
call.setTargetServiceAddress(servAddr);
call.setOperationName("sum");
```

```
Parameter param1 = new Parameter("num1", SimpleType. INTEGER, "48");
Parameter param2 = new Parameter("num2", SimpleType. INTEGER, "73");
Parameters params = new Parameters();
Params.add(param1);
Params.add(param2);
int result = ((Integer)call.invoke(params)).intValue();
.....
```

3.2 Service-side Programming Model

JSTK not only supports remote invocation on client side, but also supports server-side deployment of a SOAP service. With JSTK's server-side APIs, developers can easily deploy a SOAP service in a Servlet [8] container although JSTK is not limited and bound to Servlet container. JSTK's server-side APIs can be applied to a variety of server environments such as Servlet, JSP [9] and EJB [10] as long as it is able to retrieve SOAP request message. JSTK supports two types of Java components, Java class and EJB, to be deployed on server side. JSTK's Java class provider is specified in the class "JavaClassProvider" as follows:

```
public class JavaClassProvider extends Provider {
    public JavaClassProvider(String classPath);
    public void setClassPath(String classPath);
    public String getClassPath();
    public String process(String requestMessage) throws SoapException;
}
```

The "process" method receives and processes SOAP request message, and returns SOAP response message. When deploying a Java class, developers must provide a full class path of deployed Java class and server environment is able to access the class via this class path. The following code segment shows how to deploy a Java class on server side by using the "JavaClassProvider".

Example:

Suppose a Java class "ca.concordia.jstk.examples.Echo.java" is about to be deployed as a SOAP service, the following code segment may be used:

```
........
JavaClassProvider provider = new JavaClassProvider();
Provider.setClassPath("ca.concordia.jstk.examples.Echo");
String soapResponseMessage = provider.process(soapRequestContent);
........
```

JSTK also can deploy an EJB as SOAP service by using the class "EJBProvider" which is depicted as follows:

```
public class EJBProvider extends Provider {
    public EJBProvider(String contextProviderURL, String contextFactory,
            String providerJndiName, String providerHomeClassFullName);
    public void setContextProviderURL(String contextProviderURL);
    public String getContextProviderURL();
    public void setContextFactory(String contextFactory);
    public String getContextFactory();
    public void setProviderJndiName(String providerJndiName);
```

```
        public String getProviderJndiName();
        public void setProviderHomeClassFullName(
                String providerHomeClassFullName);
        public String getProviderHomeClassFullName();
        public String process(String requestMessage) throws SoapException;
}
```

Since JSTK is independent from a specific EJB container vendor, developers should specify the concrete context of a specific EJB container when deploying an EJB component as SOAP service. In concrete terms, EJB context information consist of two data related to a specific EJB container: EJB context provider URL, EJB context factory.

The other two contexts related to a specific EJB component are EJB JNDI name and EJB home class full name. In order to easily integrate with the popular Java application servers (BEA's WebLogic and IBM's WebSphere), JSTK uses a class called "ProviderVendor" to describe the EJB context information for WebLogic and WebSphere. The class "ProviderVendor" is defined as follows:

```
public class ProviderVendor {
  public static class WebLogic {
        public static String DEFAULT_CONTEXT_PROVIDER_URL =
                "t3://localhost:7001";
        public static String CONTEXT_FACTORY =
                "weblogic.jndi.WLInitialContextFactory";
  }
  public static class WebSphere {
```

```
        public static String DEFAULT_CONTEXT_PROVIDER_URL =
                "iiop://localhost:900";
        public static String CONTEXT_FACTORY =
                "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
    }
}
```

The following example shows how to deploy an EJB component on server side by using the "EJBProvider".

Example:

Suppose an EJB component has been deployed in WebLogic application server with default WebLogic server settings. The EJB's JNDI name is "statefulSession.TraderHome" and its home class full name is "ca.concordia.jstk.examples.statefulSession.TraderHome". The following code snippet is used to deploy this EJB component as SOAP service:

```
        .....
        EJBProvider provider = new EJBProvider();
        provider.setContextFactory(ProviderVendor.WebLogic.CONTEXT_FACTORY);
        provider.setContextProviderURL(
                ProviderVendor.WebLogic.DEFAULT_CONTEXT_PROVIDER_URL);
        provider.setProviderJndiName("statefulSession.TraderHome");
        provider.setProviderHomeClassFullName(
                "examples.ejb.basic.statefulSession.TraderHome");
        String soapResponseMessage = provider.process(soapRequestContent);
        .....
```

The following code segment is about to be used if the above EJB component is deployed in WebSphere application server with default server settings:

```
.......
EJBProvider provider = new EJBProvider();
provider.setContextFactory(ProviderVendor.WebSphere.CONTEXT_FACTORY);
provider.setContextProviderURL(
        ProviderVendor.WebSphere.DEFAULT_CONTEXT_PROVIDER_URL);
provider.setProviderJndiName("statefulSession.TraderHome");
provider.setProviderHomeClassFullName(
        "examples.ejb.basic.statefulSession.TraderHome");
String soapResponseMessage = provider.process(soapRequestContent);
.......
```

Chapter 4

Toolkit Architecture

This chapter specifies the architecture and core packages design in the JSTK toolkit, and how the design makes JSTK's architecture more flexible. The overall JSTK's design is depicted in Unified Modeling Language (UML) [11], and might use some architectural [12] and design patterns [13].

4.1 Architecture Overview

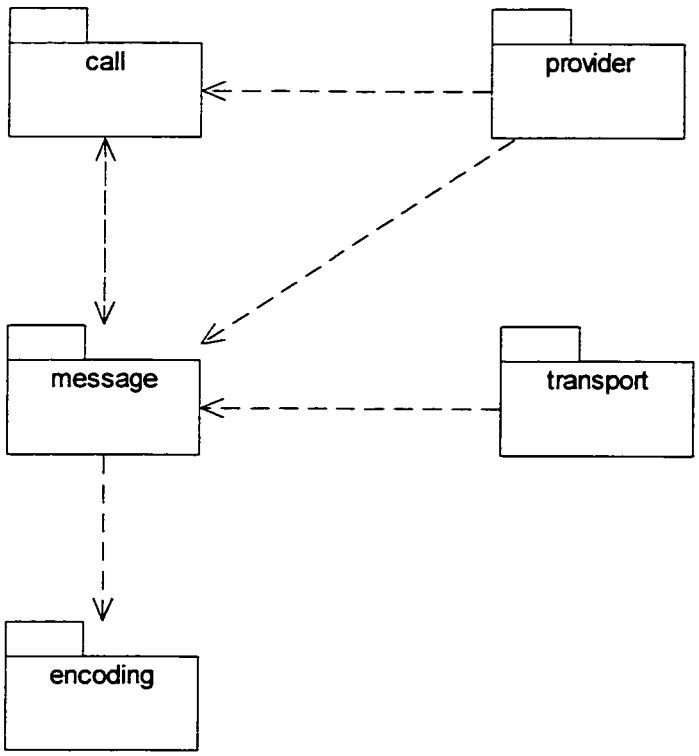


Figure 4: The Architecture of JSTK Toolkit

JSTK is composed of five core components: dynamic invocation interface in the package "call", SOAP message processor in the package "message", SOAP service provider in the package "provider", underlying transport protocol in the package "transport", SOAP

encoding and decoding in the package "encoding". The dynamic invocation interface (call) is used to collect call-related information like call service address, call name, call parameters etc. The invocation interface is a client-side programming interface for SOAP application developers. The SOAP message processor (message) is responsible for constructing and interpreting a SOAP message. The SOAP service provider (provider) is responsible for deploying a Java component (e.g. a Java class or an EJB) on server side. The underlying transport protocol (transport) implements the transport protocols to be bound to SOAP. The SOAP encoding and decoding (encoding) is used to encode/decode SOAP data types like call parameters and call return. As one of two major use scenarios of JSTK, the following diagram shows the work flows of making a client-side invocation:

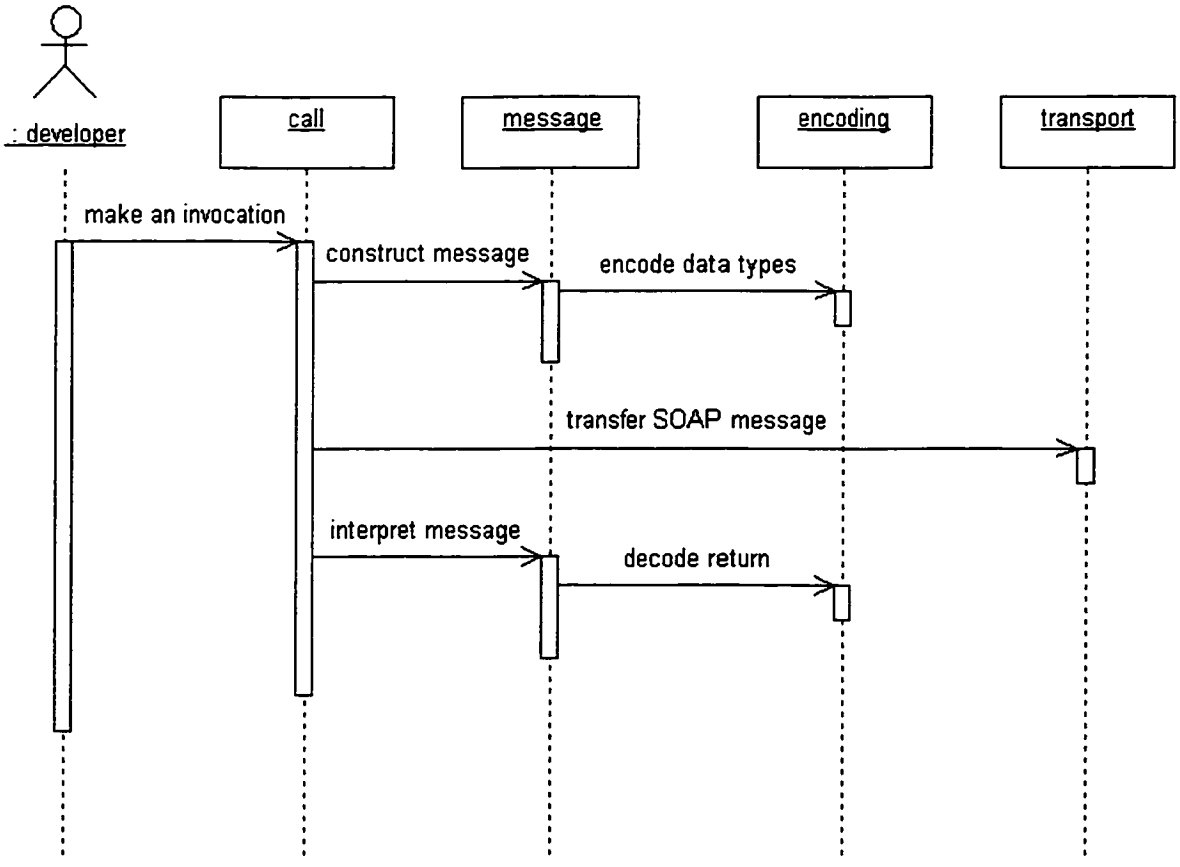


Figure 5: Sequence Diagram of Making an Invocation on Client Side

The following diagram shows the work flows of processing a SOAP request on server side:

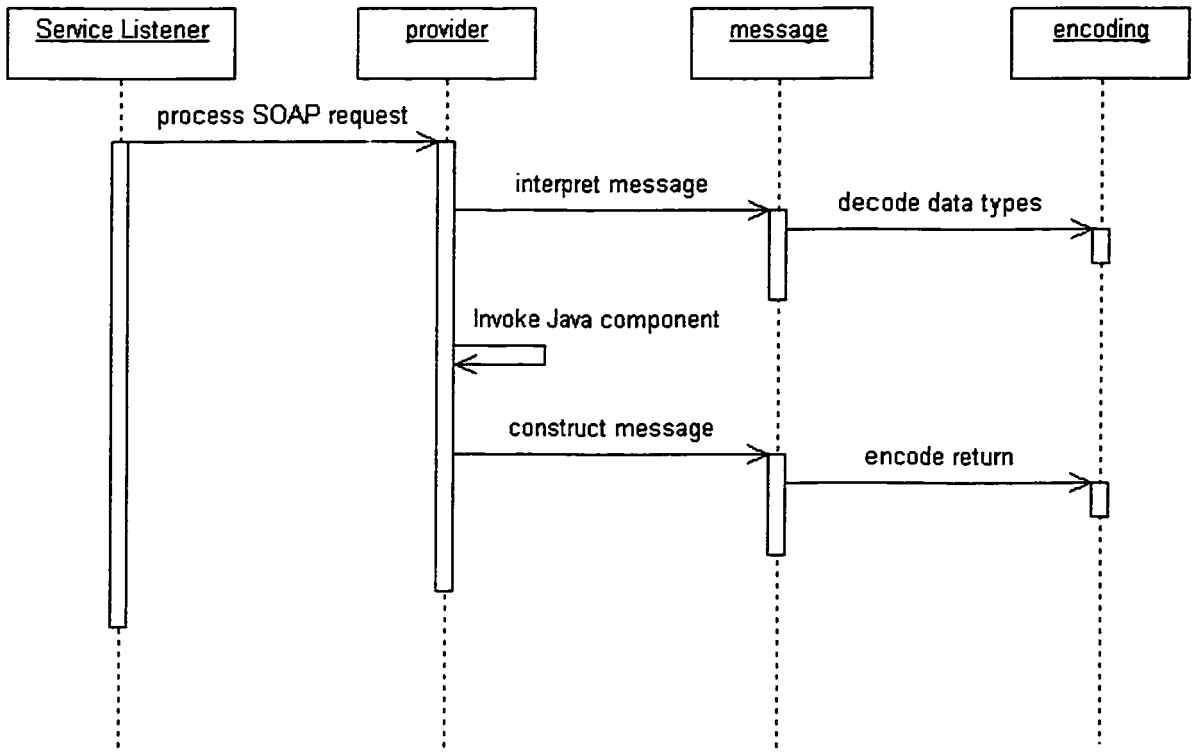


Figure 6: Sequence Diagram of Processing a SOAP Request on Server Side

4.2 Core Packages Design

4.2.1 Call Package

The call package is responsible for providing a dynamic invocation interface for developers to invoke a SOAP service on client side, and it can also be used to store a call information on server side after SOAP request has been interpreted. The call package is designed to mainly consist of four classes: Call, Parameters, Parameter and SimpleType. The Call class is used to collect and store all call-related information. The Parameters class represents a

set of parameters for a call. The Parameter class represents a parameter for a call. The SimpleType class represents a set of predefined SOAP simple types. The relationship between these classes is depicted as follows:

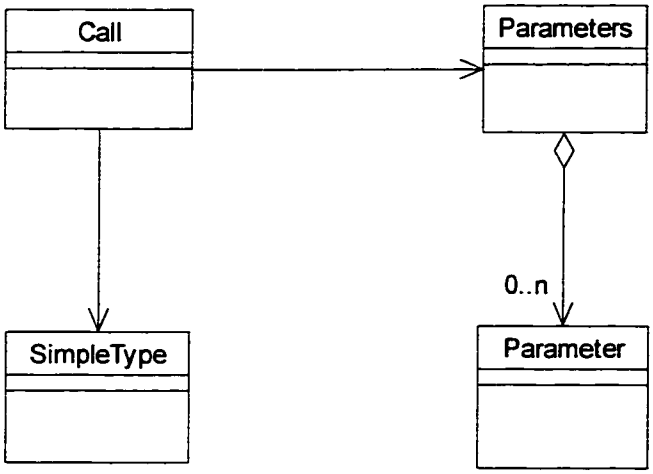


Figure 7: Class Diagram of the Call Package

4.2.2 Encoding Package

The encoding package is responsible for encoding and decoding SOAP data types. The encoding part is designed as below:

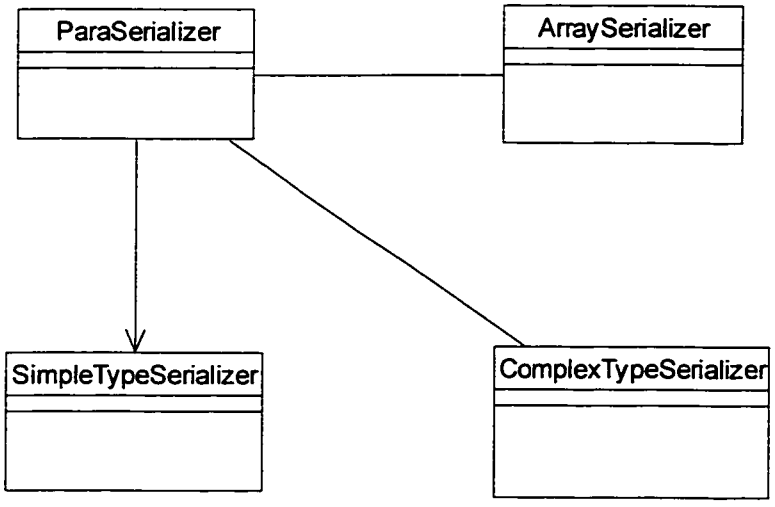


Figure 8: Encoding Class Diagram

The ParaSerializer is designed to serialize a Java data type. Based on Java data type, the ParaSerializer may use SimpleTypeSerializer to serialize a Java simple type, the ArraySerializer to serialize a Java array, the ComplexTypeSerializer to serialize a Java class of SOAP struct. Besides encoding, the encoding package is also designed to decode SOAP data types. The decoding part is designed as below:

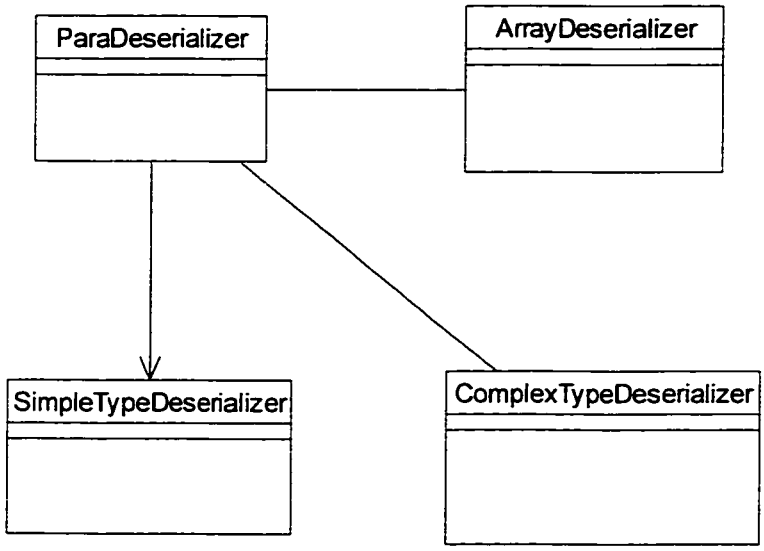


Figure 9: Decoding Class Diagram

4.2.3 Message Package

The message package is responsible for constructing and interpreting a SOAP message. The package is mainly composed of six classes: Request for SOAP request, Response for SOAP response, Message for SOAP message, Envelope for SOAP envelope in message, Body for SOAP body in envelope, Header for SOAP header in envelope. Each one of these six classes has a capability of marshalling and unmarshalling. The relationship between these classes is depicted in the following class diagram:

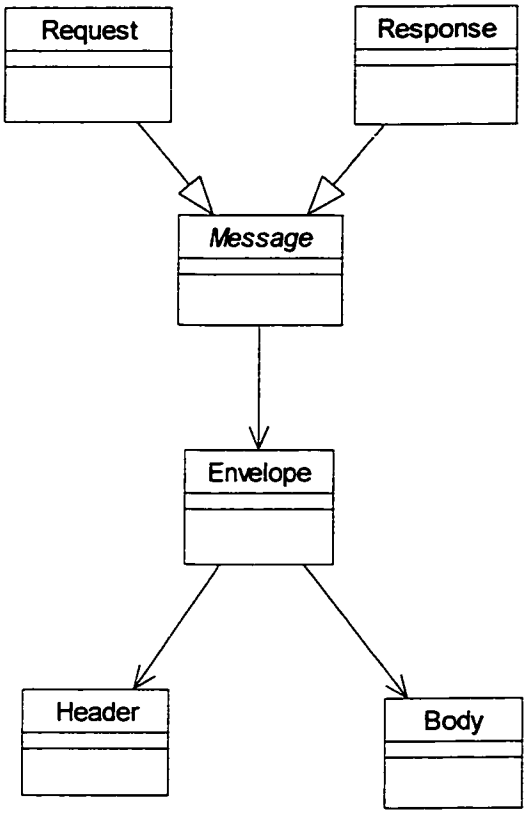


Figure 10: Class Diagram of the Message Package

4.2.4 Transport Package

The transport package is responsible for implementing the underlying protocols to be bound to SOAP. The package is designed as follow:

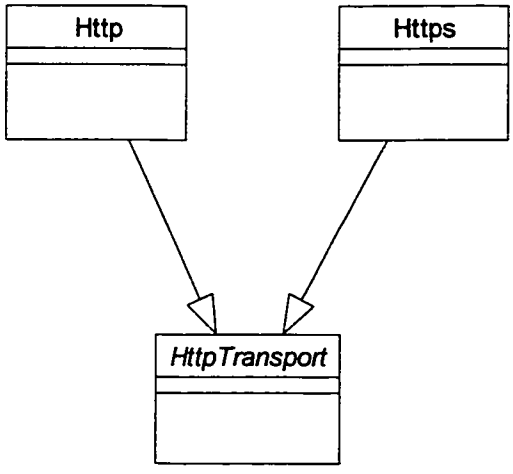


Figure 11: Class Diagram of the Transport Package

The HttpTransport is an abstract class for implementing HTTP protocol. The Http class is designed to transfer messages via HTTP. The Https class is designed to transfer message via HTTP over SSL [14].

4.2.5 Provider Package

The provider package is responsible for deploying Java component on server side. The package is designed as follows:

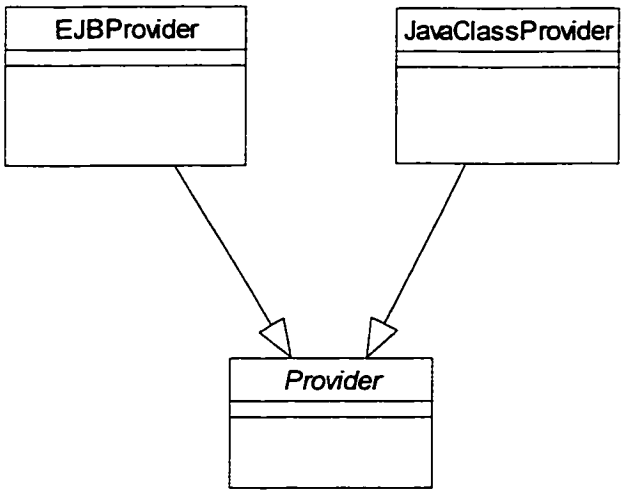


Figure 12: Class Diagram of the Provider Package

The EJBProvider is used to deploy an EJB on server side. The JavaClassProvider is used to deploy an ordinary Java class on server side. The Provider provides a common interface for all Java component providers (EJB and Java Class).

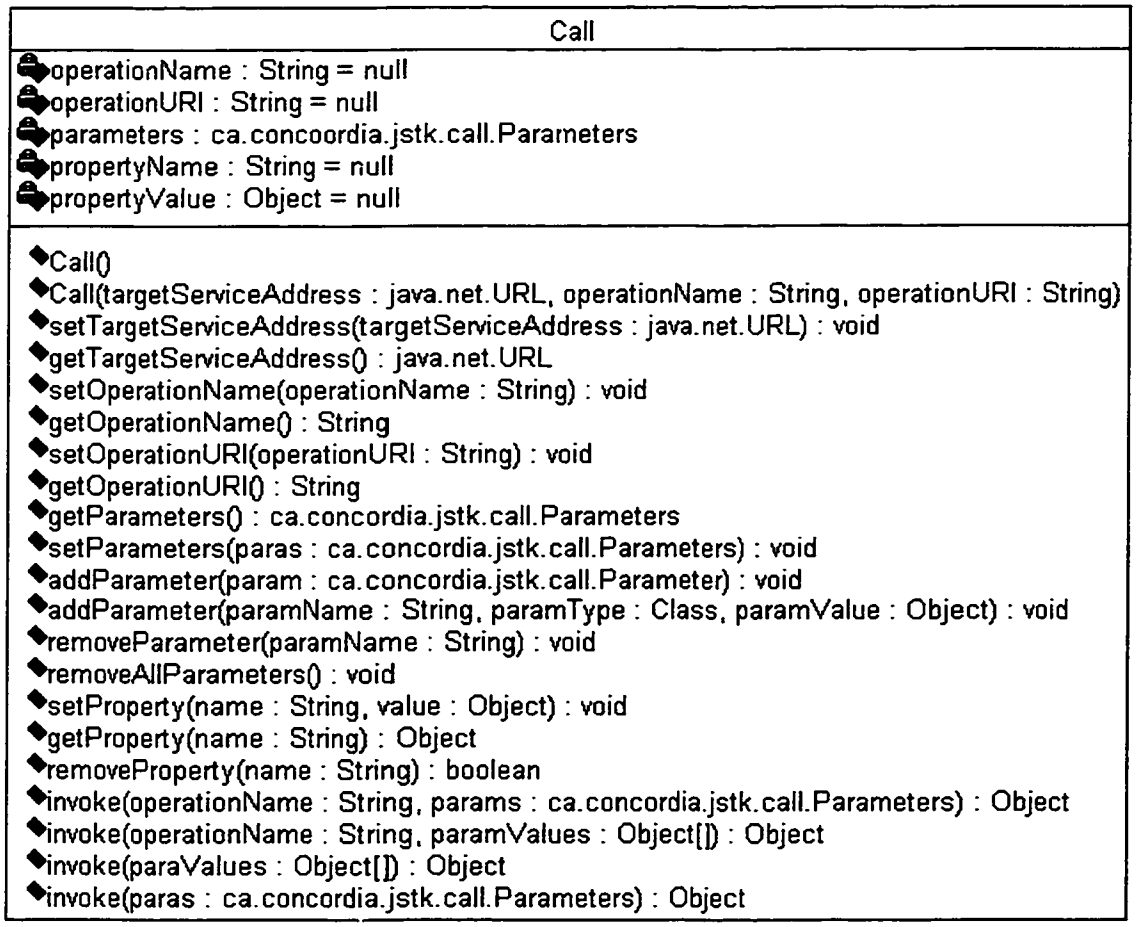
Chapter 5

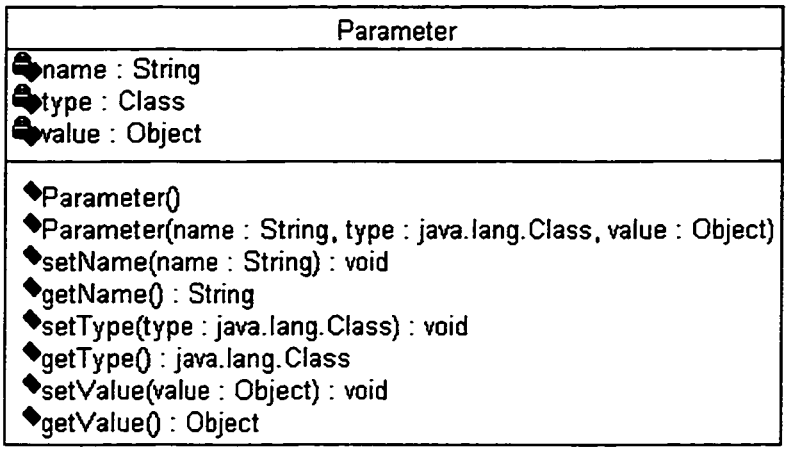
Core Packages Implementation

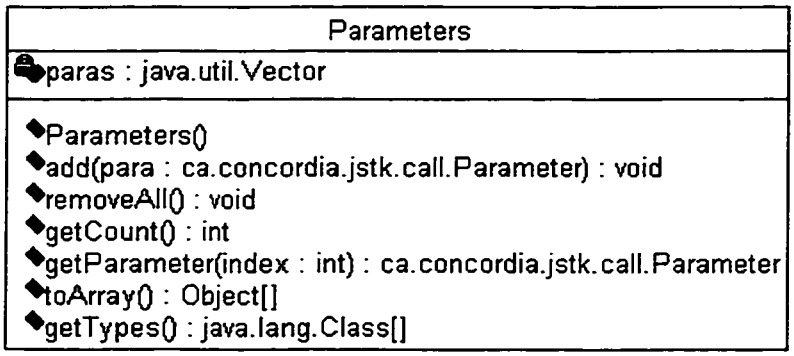
This chapter describes the implementation of JSTK's core packages: call, encoding, message, transport and provider. The classes in all packages are depicted in class diagrams.

5.1 Classes In Call Package

According to the design in chapter 4, the classes in call package are implemented as follows:

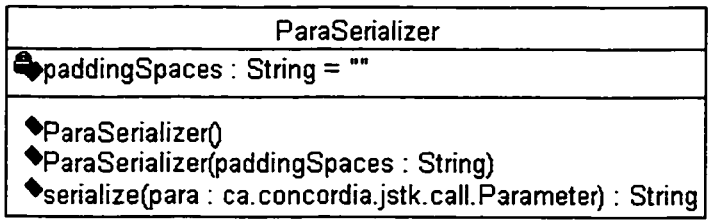


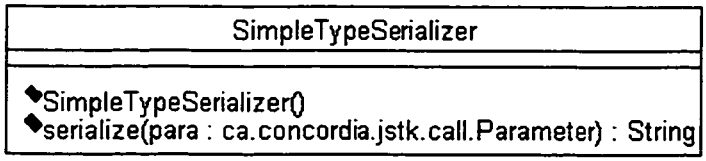


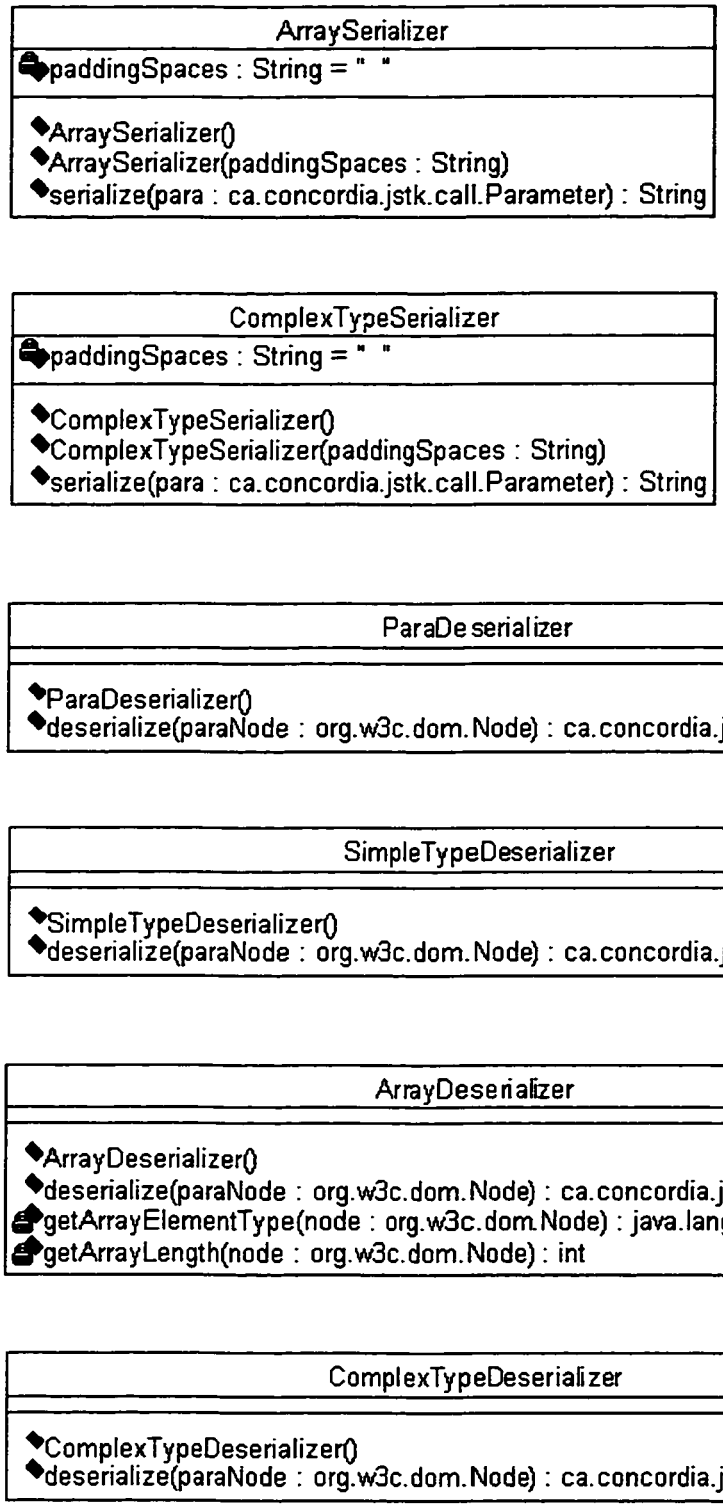


5.2 Classes In Encoding Package

According to the design in chapter 4, the classes in encoding package are implemented as follows:

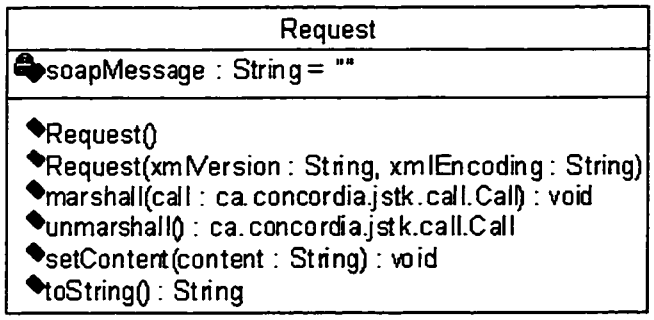


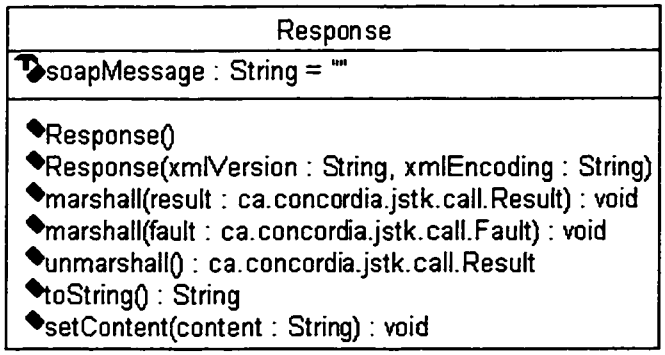


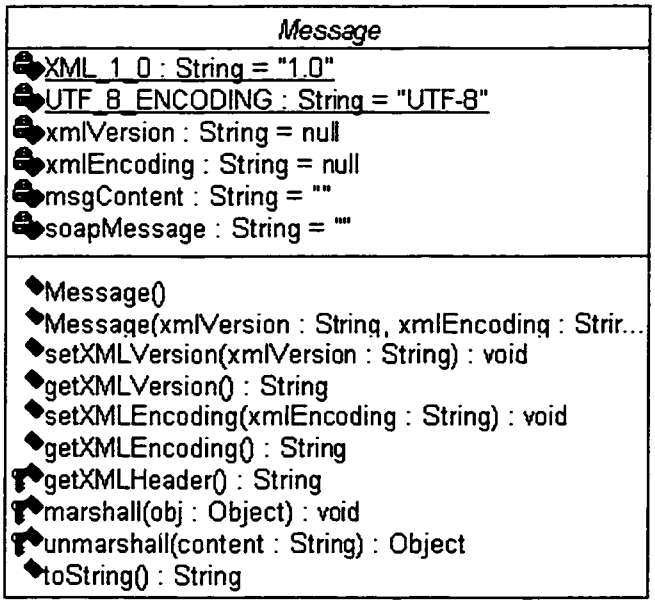

ArraySerializer
paddingSpaces : String = " "
ArraySerializer()
ArraySerializer(paddingSpaces : String)
serialize(para : ca.concordia.jstk.call.Parameter) : String
ComplexTypeSerializer
paddingSpaces : String = " "
ComplexTypeSerializer()
ComplexTypeSerializer(paddingSpaces : String)
serialize(para : ca.concordia.jstk.call.Parameter) : String
ParaDeserializer
ParaDeserializer()
deserialize(paraNode : org.w3c.dom.Node) : ca.concordia.jstk.call.Parameter
SimpleTypeDeserializer
SimpleTypeDeserializer()
deserialize(paraNode : org.w3c.dom.Node) : ca.concordia.jstk.call.Parameter
ArrayDeserializer
ArrayDeserializer()
deserialize(paraNode : org.w3c.dom.Node) : ca.concordia.jstk.call.Parameter
getArrayElementType(node : org.w3c.dom.Node) : java.lang.Class
getArrayLength(node : org.w3c.dom.Node) : int
ComplexTypeDeserializer
ComplexTypeDeserializer()
deserialize(paraNode : org.w3c.dom.Node) : ca.concordia.jstk.call.Parameter

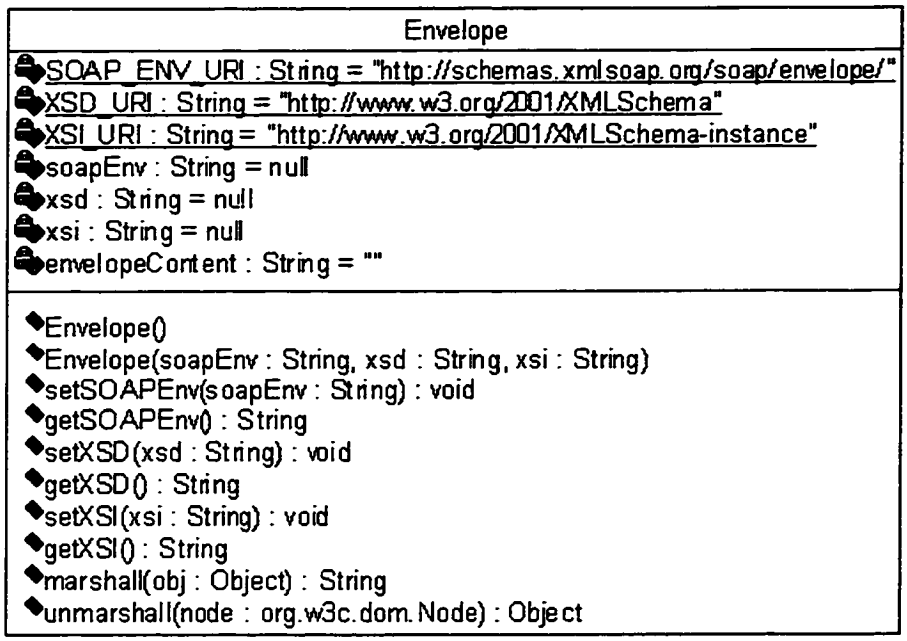
5.3 Classes In Message Package

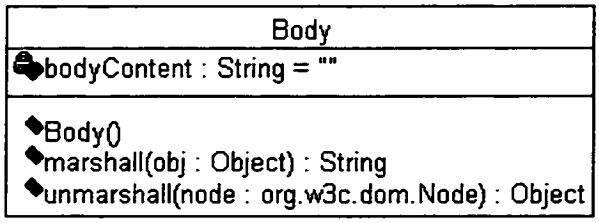
According to the design in chapter 4, the classes in message package are implemented as follows:

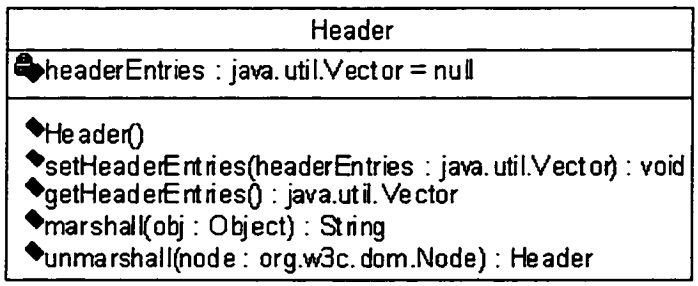






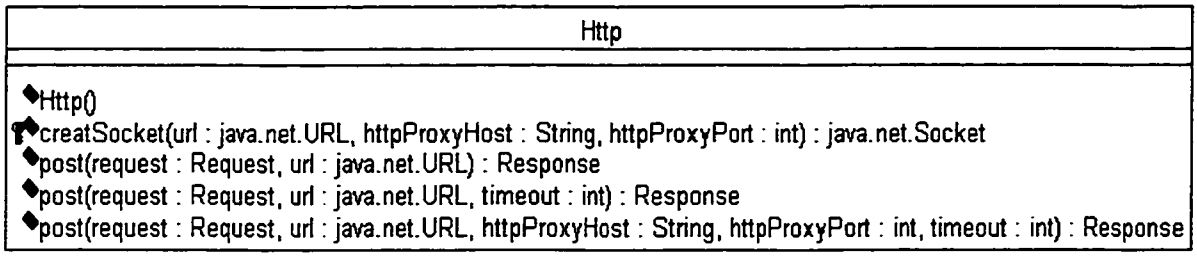

Envelope
SOAP_ENV_URI : String = "http://schemas.xmlsoap.org/soap/envelope/"
XSD_URI : String = "http://www.w3.org/2001/XMLSchema"
XSI_URI : String = "http://www.w3.org/2001/XMLSchema-instance"
soapEnv : String = null
xsd : String = null
xsi : String = null
envelopeContent : String = ""
Envelope()
Envelope(soapEnv : String, xsd : String, xsi : String)
setSOAPEnv(soapEnv : String) : void
getSOAPEnv() : String
setXSD(xsd : String) : void
getXSD() : String
setXSI(xsi : String) : void
getXSI() : String
marshall(obj : Object) : String
unmarshall(node : org.w3c.dom.Node) : Object

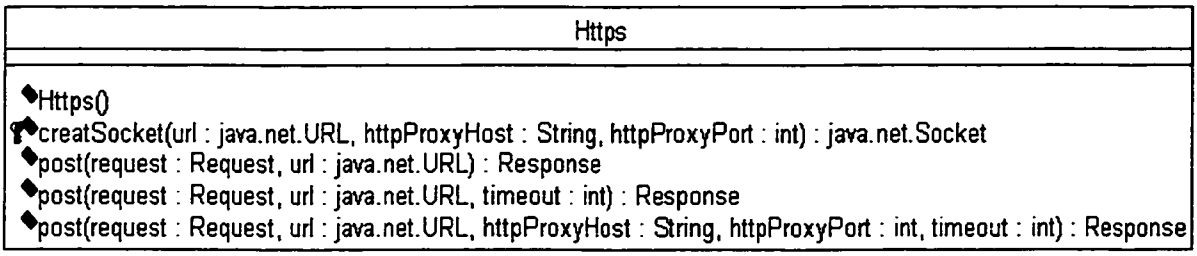

Body
bodyContent : String = ""
Body()
marshall(obj : Object) : String
unmarshall(node : org.w3c.dom.Node) : Object

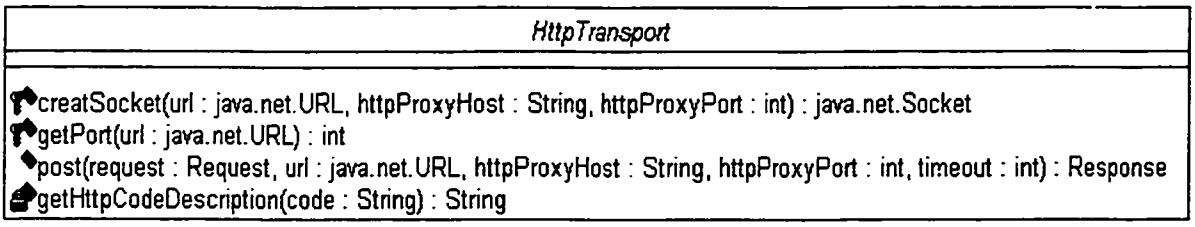

Header
headerEntries : java.util.Vector = null
Header()
setHeaderEntries(headerEntries : java.util.Vector) : void
getHeaderEntries() : java.util.Vector
marshall(obj : Object) : String
unmarshall(node : org.w3c.dom.Node) : Header

5.4 Classes In Transport Package

According to the design in chapter 4, the classes in transport package are implemented as follows:

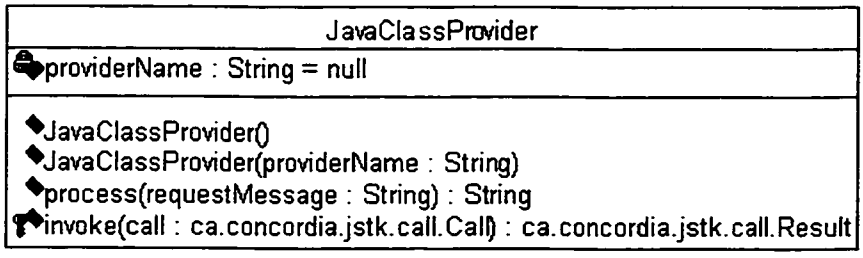


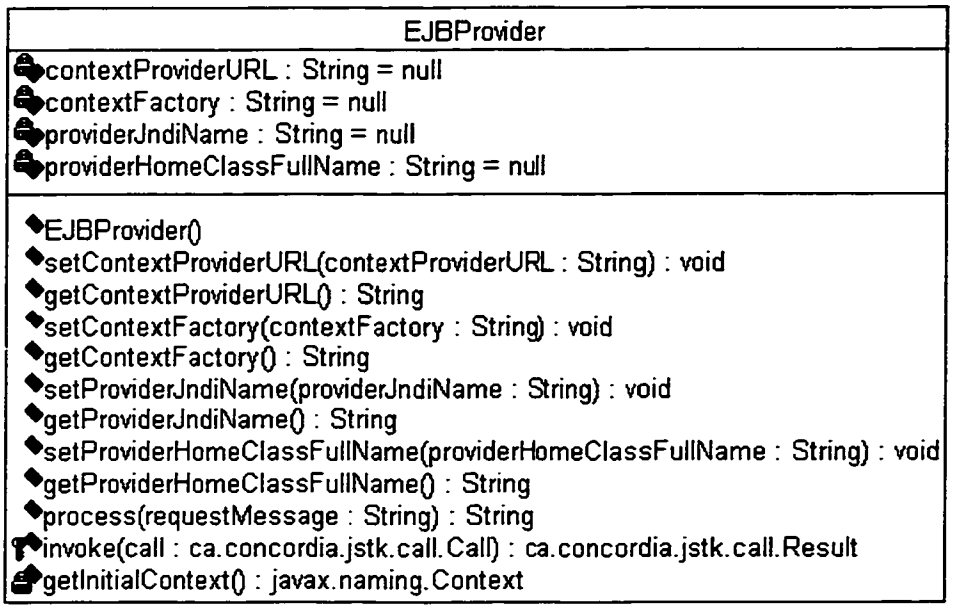


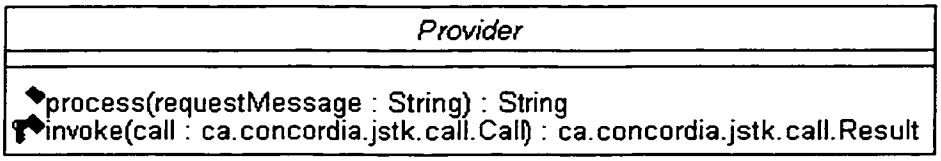


5.5 Classes In Provider package

According to the design in chapter 4, the classes in provider package are implemented as follows:




EJBProvider
contextProviderURL : String = null
contextFactory : String = null
providerJndiName : String = null
providerHomeClassFullName : String = null
EJBProvider()
setContextProviderURL(contextProviderURL : String) : void
getContextProviderURL() : String
setContextFactory(contextFactory : String) : void
getContextFactory() : String
setProviderJndiName(providerJndiName : String) : void
getProviderJndiName() : String
setProviderHomeClassFullName(providerHomeClassFullName : String) : void
getProviderHomeClassFullName() : String
process(requestMessage : String) : String
invoke(call : ca.concordia.jstk.call.Call) : ca.concordia.jstk.call.Result
getInitialContext() : javax.naming.Context


Provider
process(requestMessage : String) : String
invoke(call : ca.concordia.jstk.call.Call) : ca.concordia.jstk.call.Result

Chapter 6

Conclusion and Future Work

6.1 Conclusion

As a light-weight Java implementation of SOAP 1.1, JSTK provides developers a series of simplified APIs to access and deploy a SOAP service on both client and server side. With JSTK, developers not only can quickly transform an ordinary Java class or an EJB component into a SOAP service, but can easily invoke an existing SOAP service by using JSTK's simplified dynamic invocation interface. In addition, JSTK supports a variety of data types and integrates with the most popular Java application servers (BEA's WebLogic and IBM's WebSphere), which makes it possible that developers employ JSTK to develop various real-world software applications. In concrete terms, JSTK has the features as follows:

- SOAP Specification 1.1 Conformance
- Wealth of SOAP Data Types and Nesting Support
- Support of Java Class and EJB component as SOAP Service
- Support of SOAP/HTTP and SOAP/HTTPS Bindings
- Intergrated With WebLogic and WebSphere
- Simple APIs for Both Client Side and Server Side

Moreover, JSTK adopts a flexible architecture, making its functionality extension much easier.

6.2 Future Work

As the SOAP specifications are constantly being updated and Web Services technology is evolving, JSTK is planned to extend its functionality to fully support the latest SOAP and Web Services specifications in the near future. The future features of JSTK are depicted as follows:

- Conform to SOAP Specification 1.2
- Support Generated Client-side Stub From WSDL [15]
- Support WSDL/Java Mapping
- Support SOAP Message With Attachment
- Support SOAP/SMTP Binding
- Support One-way Invocation Method

In addition, JSTK might redesign and implement the encoding package in order to improve its encoding/decoding efficiency.

References

[1] Daniel Austin, Abbie Barbir, Christopher Ferris, Sharad Garg, Web Services Architecture Requirements, World Wide Web Consortium (W3C), 2002

[2] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Extensive Markup Language(XML) 1.0 (Second Edition), World Wide Web Consortium (W3C), 2000

[3] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, Dave Winer, Simple Object Access Protocol (SOAP) Specification 1.1, World Wide Web Consortium (W3C), 2000

[4] R. Fielding , J. Gettys , J. Mogul, H. Frystyk , L. Masinter , P. Leach T. Berners-Lee, Hypertext Transfer Protocol -- HTTP/1.1, Network Working Group(RFC2616), 1999

[5] Mark Baker, Martin Gudgin, Oisín Hurley, Marc Hadley, John Ibbotson, Scott Isaacson, Yves Lafon, Jean-Jacques Moreau, Henrik Frystk Nielsen, Krishna Sankar, Nick Smilonich, Lynne Thompson, XML Protocol Abstract Model, World Wide Web Consortium (W3C), 2001

[6] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Java Language Specification (Second Edition), Sun Microsystems Inc., 2000

[7] Paul V. Biron, Ashok Malhotra, XML Schema Part 2: Datatypes, World Wide Web Consortium (W3C), 2001

[8] Danny Coward, Java™ Servlet Specification Version 2.3, Sun Microsystems Inc., 2001

[9] Mark Roth, Eduardo Pelegrí-Llopart, JavaServer Pages(TM) Specification 2.0, Sun Microsystems Inc., 2001

[10] Linda G. DeMichiel, L. Ümit Yalçinalp, Sanjeev Krishnan, Enterprise JavaBeans Specification(Version 2.0), , Sun Microsystems Inc., 2001

[11] G. Booch, J. Rumbaugh, I. Jocobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999

[12] Mary Shaw, David Garlan, Software Architecture – Perspectives on An Emerging Discipline, Prentice Hall, 1997

[13] E. Gamma, R. Helm, R. Johnson , J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addsion-Wesley, 1994

[14] Alan O. Freier , Philip Karlton , Paul C. Kocher, The SSL Protocol Version 3.0, Transport Layer Security Working Group, 1996

[15] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, Web Services Description Language (WSDL) 1.1, World Wide Web Consortium (W3C), 2001